UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E ENGENHARIA DE COMPUTAÇÃO

GRUPO DELTA
ÁTILA DA ROCHA COSTA E SILVA
BÉUREN FELIPE BECHLIN
FELIPE BERTOLDO COLOMBO DE SOUZA

Implementação do jogo War sando TypeScript

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Lucas Mello Schnorr
Orientador

Porto Alegre
2018

SUMÁRIO

1 INTRODUÇÃO

O trabalho visa a solução de um problema através de diferentes paradigmas de programação: orientação a objetos e funcional.

Com as duas implementações, teremos visto na prática as vantagens e desvantagens de cada paradigma na abordagem ao problema, tendo condições de estabelecer um comparativo entre diferentes pontos.

O problema escolhido foi o jogo War, versão brasileira do norte americano Risk, que será abordado a seguir.

1.1 Visão geral da linguagem

Dentro das linguagens disponibilizadas, TypeScript é a que possui maior suporte para elementos de interfaces e bibliotecas em geral, já que é um superset de JavaScript que, através do TypeScript Transpiler, é transformada para JavaScript. Outro fator determinante é a necessidade de solucionar o problema também de uma forma funcional, e JavaScript é uma linguagem essencialmente funcional com vasta utilização de closures e padrões de projeto, como módulo.

Mesmo TypeScript sendo implementada para restringir o uso de JavaScript, além de adicionar tipagem estática e outros elementos de OOP, é possível utilizar esses padrões herdados da linguagem JavaScript, ajudando bastante em uma solução funcional. Como já citado, o TypeScript foi desenvolvido com o intuito de criar uma linguagem mais estável para aplicações web e com melhor manutenibilidade que o JS. Adicionando elementos de checagem estática de tipos, diferentemente do JS, orientação a objetos com maior poder que as especificações de EcmaScript 6.

Dessa maneira, possuímos uma linguagem completa para solucionar um problema com grande interação com usuário e elementos gráficos como é um caso de um jogo. Para exemplificar, é possível utilizar o framework de single page applications criado pelo Google chamado Angular, usado por grandes empresas. Também é possível utilizar uma biblioteca interface com usuário chamada React, desenvolvida e mantida pelo Facebook usada por Netflix, AirBnb e outros.

Para auxílio da programação funcional existe a implementação da biblioteca underscore para TypeScript. Além disso a linguagem possui um ótimo ambiente de desenvolvimento com vários utilitários desenvolvidos por terceiros para gerenciamento de

pacotes e dependências, gerenciadores de tarefas e frameworks para testes.

1.2 Descrição do problema

Em TypeScript será implementado o jogo War, versão brasileira do norteamericano Risk.

No jogo, o participante controla um exército que recebe um objetivo que deverá ser cumprido. O objetivo é conquistar determinado território. Se, ao final de uma batalha, o participante destruir todos os exércitos de defesa do adversário, ele terá conquistado o território. Quando um objetivo é declarado como concluído, o participante que o declarou conquista o território de combate e recebe novos exércitos, aumentando seu poder.

Em cada rodada, o participante poderá receber novos exércitos, caso conquiste territórios, usá-los conforme sua estratégia, atacar outros exércitos ou mover exércitos.

Vence o participante que atingir o objetivo recebido no início do jogo.

2 DESENVOLVIMENTO

2.1 Dependências

A única dependência que deve estar instalada no host para o processo de build do projeto é o **NodeJS**. O NodeJS é uma versão da V8, engine de JavaScript desenvolvida pelo Google para pré-compilação e execução de JavaScript em seu navegador Google Chrome, portada para plataforma.

Um dos pontos importantes dessa engine de JavaScript é que ela é totalmente single thread, mas usa um modelo de eventos e I/O não bloqueante para lidar com essa limitação. Esse modelo tem se mostrado uma alternativa aos tradicionais com multi threads, tanto que o NodeJS vem sendo vastamente usado como servidor aplicação para aplicações web.

Juntamente com o NodeJS, é instalado o gerenciador de pacotes **NPM**, que possui o maior ecosistema de bibliotecas open source do mundo. Essas dependências são instaladas localmente dentro de uma pasta na mesma raiz do projeto, independente da plataforma e sem nenhuma ligação com sistema operacional do usuário. Isso aumenta a portabilidade do projeto e automatiza o processo de construção da aplicação, já que com um comando é baixado e instalado todas as dependências listadas no projeto.

2.1.1 Instalando a aplicação

Como citado anteriormente o projeto usa o **NodeJS** e **NPM**. No primeiro momento é necessário instalar as dependências externas do projeto, como até mesmo o **Typescript**, **Babel transpiler**, **Redux**, **React**, **MobX** e outros.

Para realizar a instalação, execute via terminal o seguinte comando no diretório raíz do projeto:

```
1  npm install
```

Esse comando chama o gerenciador de pacotes **NPM** solicitando a instalação dos pacotes que estão listados no arquivo `PACKAGE.json`. Após instaladas as dependências, é necessário "compilar"o projeto, nas próximas sessões será descrito como ocorre esse processo.

Para iniciar o ambiente de desenvolvimento é necessário executar o seguinte co-

mando:

```
1  npm run start
```

Esse comando chama o script start também descrito no arquivo `PACKAGE.json`. Como resultado, ele compila o projeto, abre o navegador padrão do host (apontando para o localhost porta 3000), e já sobe um servidor de aplicação node que serve o conteúdo estático ligado com todas as interfaces de rede do Host na porta 3000. Além disso, é adicionado um watcher na pasta, com Inotify em linux distros, para recompilar o projeto em casos de modificações e automaticamente recarregar a página do navegador.

2.1.2 React

React é a biblioteca usada para a interface gráfica, tendo influência na construção da aplicação. Criada pelo Facebook, visa solucionar um problema antigo relacionado à construção e manipulação da DOM e a lógica de negócios da aplicação. Com a evolução da computação em geral e também dos requisitos dos usuários, as páginas Web passaram de simples páginas com hiperlinks para aplicações completas e, com isso, o browser passa a ser usado como um ambiente de execução dessas aplicações.

A biblioteca tem uma construção como teor funcional, no qual cada componente tem entradas via propriedades e retorna elementos HTML através de JSX. Os componentes podem montar outros componentes em sua visualização e definir suas entradas. É possível, também, armazenar estados dentro de componentes por meio de classes. Para isso, estende-se uma classe de componente do React, que possui alguns métodos adicionais que lidam com o estado e também com o ciclo de vida do componente.

2.1.3 Redux

O conjunto React+Redux é uma combinação comum para aplicações React. Usar o estado do componente para salvar dados e estado geral da aplicação não é muito escalável, já que se torna necessário ter componentes com muito filhos e muitas propriedades de entrada para componentes intermediários. Então, a combinação do React com um gerenciador de estado mais completo é quase que natural.

O Redux é um gerenciador e mantenedor de estado para aplicações JavaScript em geral. Seu principal conceito está em guardar o estado de forma imutável e sempre que

seja necessário modificar o estado gerar um novo objeto. Essa decisão é muito importante, pois em geral todas as comparações realizadas em JavaScript são da forma shallow, e assim gerando um novo objeto é fácil compreender quando houve alguma alteração. Pela forma que a linguagem JavaScript foi concebida, usando cadeia de protótipos, uma operação de comparação deep é bem mais custosa já que é necessário percorrer toda a árvore de protótipos existentes em um objeto.

Para guardar esse objeto que contém os dados de estado é introduzido um conceito chamado **store** que possui duas interfaces principais: uma para pegar o estado e outra para disparar ações. As ações são objetos que contêm basicamente o tipo da ação que está disparando uma carga útil, que é usado como parâmetro para atualizar o estado. Essa é a única maneira de sinalizar uma alteração de estado.

A real alteração do estado é realizada dentro de **redutores**, que são basicamente funções que detalham o que cada ação deve gerar como resultado. Em geral, são criados com switchs com a variável seletora no tipo da ação e devem retornar um novo objeto que será substituído para representar o novo estado.

A real utilidade Redux no caso do React é uma função de alta ordem disponível na pacote react-redux, onde é possível conectar os componentes React com a store. Nessa conexão com os componentes são injetados alguns atributos disponíveis na store como propriedades do componente React, através de uma função de mapeamento entre o estado e as propriedades. Essa função de mapeamento deve ser uma função pura, podendo ser realizado algumas manipulações antes de retornar as propriedades.

2.1.4 MobX

O conjunto React+MobX é uma combinação não tão comum para aplicações React. Assim como o Redux, MobX serve para controlar o estado da aplicação fora dos componentes React garantindo a escalabilidade e manutenibilidade do projeto, mas ao contrario da sua alternativa funcional Redux, MobX trabalha muito bem com o paradigma orientado a objetos, integrando as funcionalidades tanto de ações e redutores em classes, o que acaba convendo uma clareza e familiraidade não vista antes na etapa funcional do projeto usando sua alternativa, Redux. O cerne do MobX está no uso dos decoradores **@observer** e **@observable**, **@observable** encapsula um objeto, array ou instancia de classe permitindo que o MobX tenha visão sobre o uso do seu respectivo **setter**, esse acesso especial permite framework reaja as mudanças em dados marcados como **@ob-**

servable. Para permitir as reaçoes dos componentes **React** usamos o decorador **@observers**, ao encapsular os componentes marcados ele força a chamada do método **render** do componente toda vez que algum dado marcado como **@observable** acessado dento do componente mude seu valor. Essa capacidade junto com uma série de otimizacões do framework permitem que a UI reaja rapidamente a mudanças no estado sem prejudicar a legibilidade do código.

@action e @computed

2.2 TypeScript vs. JavaScript

Ao pé da letra JavaScript não é uma linguagem, mas sim uma definição feita pela ECMA foundation para servir como um contrato entre os navegadores e desenvolvedores de aplicações web sobre o que está disponível na sua engine de script. Isso nem sempre é totalmente cumprido e, com o processo de evolução da linguagem, vários navagores não implementam a especificação por completo.

Esses problemas acabaram levando a soluções alternativas pela comunidade, como implementar as funcionalidadesnovas da linguagem usando elementos disponíveis no momento. Essa metodologia ficou conhecida como polyfill. Isso também foi evoluindo, criando um conjunto grandes de polyfills e então se começou a realizar o processo de transpilação, onde o código fonte JavaScript é transformado novamente em JavaScript, usando as funcionalidades das especificações antigas.

O TypeScript, criado pela Microsoft, usa esses artifícios para gerar JavaScript, executado em grande parte dos navagadores web. Esse caso é diferente, uma vez que trata-se de uma linguagem sendo transformada em outra, mas muitas os dois processos são realizados serialmente com TSCompiler, que transforma em ES6(EcmaScript v6) e depois em ES5(EcmaScript v5), melhor suportada pela maioria dos navegadores.

Entretanto, a própria referência do TypeScript se intitula como um superset de JavaScript adicionando, principalmente, a tipagem estática, o que não a torna disruptiva em relação ao JavaScript.

Outro ponto interessante nesse contexto é a possibilidade de usar grande parte das bibliotecas disponíveis para JavaScript realizando uma interface para compatibilidade com a tipagem do TypeScript, que muitas vezes pode ser feita pelo próprio usuário da biblioteca, já que o TScompiler também compila código JavaScript. Isso pode ser visto no arquivo `package.json` onde são adicionados os módulos `@types` para baixar os

pacotes que contém essas interfaces e namespaces.

```
{
 "name": "war-game",
 "version": "0.1.0",
 "private": true,
 "homepage": "http://bfbechlin.github.io/war-game",
 "dependencies": {
   "bootstrap": "^4.1.0",
   "classnames": "^2.2.5",
   "history": "^4.7.2",
   "material-ui": "^0.20.0",
   "mobx": "^5.0.2",
   "mobx-react": "^5.2.3",
   "react": "^16.3.1",
   "react-dom": "^16.3.1",
   "react-redux": "^5.0.7",
   "react-router-redux": "^4.0.8",
   "react-scripts-ts": "2.14.0",
   "react-swipeable-views": "^0.12.13",
   "react-transition-group": "^1.2.1",
   "redux": "^3.7.2",
   "redux-devtools-extension": "^2.13.2",
   "svg-path-parser": "^1.1.0",
   "underscore": "^1.9.0",
   "uuid": "^3.2.1"
 },
 "scripts": {
   "start": "react-scripts-ts start",
   "build": "react-scripts-ts build",
   "test": "react-scripts-ts test --env=jsdom",
   "eject": "react-scripts-ts eject",
   "predeploy": "npm run build",
   "deploy": "gh-pages -d build",
   "report": "pdflatex -no-file-line-error report/relatorio.tex"
 },
 "devDependencies": {
   "@types/classnames": "^2.2.3",
   "@types/jest": "^22.2.2",
   "@types/node": "^9.6.2",
   "@types/react": "^16.3.10",
   "@types/react-dom": "^16.0.5",
```

```
41    "@types/react-redux": "^5.0.16",
42    "@types/react-router-redux": "^5.0.13",
43    "@types/material-ui": "^0.21.1",
44    "@types/react-swipeable-views": "^0.12.1",
45    "@types/react-transition-group": "^1.1.3",
46    "@types/underscore": "^1.8.8",
47    "@types/uuid": "^3.4.3",
48    "gh-pages": "^1.1.0",
49    "typescript": "^2.8.1"
50  }
51 }
```

Esse processo é usado por outras linguagens além do TypeScript, como ClojureScript, CoffeeScript e outros.

2.3 Paradigma funcional

2.3.1 Estrutura do projeto

Na pasta `src` se encontra todo o fonte do projeto com a seguinte hierarquia de pastas:

- **Components**: componentes React. Separado por componentes chaves, mas que possuem outros componentes relacionados/dependentes.
- **Core**: lógica de negócio em geral. Com as seguintes subpastas:
  - **Transducers**: funções puras, como mapeamento de estado para propriedades, levantamento de possíveis países para atacar.
  - **Transitions**: funções que precisam ler o estado da aplicação para operar.
- **Hocs**: componetes de alta ordem React.
- **Store**: gerenciador de estado Redux. Separado por grupo de funcionalidades, contendo o seu redutor e suas ações.
- **Utils**: utilidades em geral, como funções de alta ordem para manipulação de listas e objetos, cores e etc.

Existem arquivos soltos dentro da pasta `src` que são os arquivos de ponto de entrada para a criação da aplicação.

2.3.2 Conceitos explorados nas funções

2.3.2.1 utils/array

Funções de alta ordem aplicáveis em arrays. O JavaScript/TypeScript por padrão possui uma grande biblioteca com funções de mais alta ordem.

Função **intersection** retorna a interseção de dois conjuntos aqui representados por listas. Faz uso da função de alta ordem `filter` para retornar os elementos em comum entre os dois arranjos. O callback esperado como parâmetro da função filter é outra função, definida ali mesmo, que estabelece a condição de que o elemento precisa estar em ambos os arranjos.

```
1 export const intersection = <T>(array1: T[], array2: T[]) => (
2  array1.filter((element: T) => array2.indexOf(element) > -1)
3 );
```

A função **difference** retorna a diferença entre dois conjuntos, nesse caso é importante a ordem já que a A - B é diferente de B - A. Assim como a intersection, utiliza filter e outra função para callback. A diferença está na condição estabelecida no callback da filter: nesse caso, os elementos que serão retornados não podem estar concomitantemente em ambos os arranjos, ou seja, calcula a diferença entre os dois.

```
1 export const difference = <T>(array1: T[], array2: T[]) => (
2  array1.filter((element: T) => array2.indexOf(element) < 0)
3 );
```

A função **incremento cíclico** retorna o próximo item do array em relação a um elemento descrito. Caso esse elemento esteja na última posição do array então é retornado o primeiro elemento.

```
1 export const cyclicIncrement = <T>(array: T[], element: T) => (
2  array[(array.indexOf(element) + 1) % array.length]
3 );
```

A função **shuffle** retorna um vetor com os elementos embaralhados do arranjo recebido como parâmetro. Para isso, faz uso de recursão onde cada chamada adiciona

um elemento aleatório do primeiro vetor no novo arranjo, parando quando o tamanho do arranjo source for igual à zero. Como efeito colateral essa função altera o próprio array enviado como argumento, já que em JS objetos e listas são sempre passados como referências.

```
export const shuffle = <T>(array: T[]): T[] => {
 if (array.length === 0) {
   return [];
 }
 const index = Math.trunc(Math.random() * array.length);
 const element = array.splice(index, 1);
 return [element[0], ...shuffle(array)];
};
```

A função **partition** retorna uma lista de n partições, cada uma contendo uma lista de itens. Faz uso da função groupBy da biblioteca underscore para agrupar os dados que retornam o mesmo valor na função de agrupamento. A função values transforma o objeto criado pelo groupBy em um lista com listas dos grupos formados.

```
export const partition = <T>(array: T[], n: number): T[][] => {
 const result = groupBy(array, (item: T, index: number) => (
   index % n
 ));
 return values(result);
};
```

2.3.2.2 utils/object

Funções de alta ordem aplicáveis em objetos. Para objetos a linguagem não provê nenhuma função de alta ordem, devido à heterogeneidade dessa construção. Nessas funções, não tivemos outra opção a não ser usar o controle de fluxo for, já que é a única forma de iterar sobre propriedades de um objeto.

Outro ponto importante aqui é que todas as propriedades do objeto tenham a mesma interface de elementos que é especificada através dos templates de TypeScript <T>.

A função **filter** filtra as propriedades de um objeto com base na função passada como argumento. Nesse função é injetado dois parâmetros o elemento contido na propriedade que se está iterando e opcionalmente o nome dessa propriedade.

```
export const filter = <T>(obj: object, callback: ((value: T, key?:
    string) => boolean)): string[] => {
const props = [];
for (const prop in obj) {
 if (obj.hasOwnProperty(prop)) {
   if (callback(obj[prop], prop)) {
     props.push(prop);
   }
 }
}
return props;
};
```

O **map** consiste no mesmo compartimento que a função tem para arrays. Onde para cada propriedade é gerado um novo elemento. Esse elemento deve possuir a interface generica <R>.

```
export const map = <T, R>(obj: object, callback: ((property: T, key?:
    string) => R)): R[] => {
const result: R[] = [];
for (const prop in obj) {
 if (obj.hasOwnProperty(prop)) {
   result.push(callback(obj[prop], prop));
 }
}
return result;
};
```

A função **lenght** conta a quantidade de propriedades de um objeto.

```
export const lenght = (obj: object): number => {
let n: number = 0;
for (const prop in obj) {
 if (obj.hasOwnProperty(prop)) {
   n += 1;
 }
}
return n;
};
```

A função **toArray** transforma um objeto em um array, fazendo com que cada propriedade desse objeto seja um item no array gerado.

```
export const toArray = <T>(obj: object): T[] => {
const result: T[] = [];
for (const prop in obj) {
 if (obj.hasOwnProperty(prop)) {
   result.push(obj[prop]);
 }
}
return result;
};
```

2.3.2.3 core/transducers/map

Para a compreensão funções dessa parte, é importante entender como o objeto de estado dos países está estruturado.

```
export interface CountryState {
 'East Africa': CountryInfo;
 'Egypt': CountryInfo;
...
}

export interface CountryInfo {
 troops: number;
 owner: string;
 hovered: boolean;
}

export type Countries = 'East Africa' | 'Egypt' | ...

export interface ContinentInfo {
 color: Color;
 troopsBonus: number;
 countries: Countries[];
}

export interface Continents {
 [index: string]: ContinentInfo;
}

export const borderCountries: BorderCountry = {
```

```
26  'Brazil': ['Peru', 'Venezuela', 'Argentina', 'North Africa'],
27  'East Africa': ['North Africa', 'Egypt', 'South Africa', 'Congo'],
28  ...
29 }
```

A função **playerCountries** utiliza a função de alta ordem filter definida acima para o fim de possibilitar um fácil acesso aos territórios que pertencem ao jogador passado como parâmetro. São passados três parâmetros, o identificador do jogador ao qual devem pertencer os objetos Country resultantes da consulta, os Countries que compõem as regiões possíveis do mapa e por fim um parâmetro opcional minTroops que serve uma função lógica para omitir Countries que tem um número menor que o valor passado como parâmetro no resultado.

```
1 export const playerCountries = (player: string, countries: CountryState
      , minTroops: number = 0) => (
2 filter(countries, (country: CountryInfo) => ( country.owner === player
      && country.troops > minTroops))
3 );
```

A função **borderCountries** recebe como parâmetro um Country, o identificador do jogador ao qual pertence, e os Countries que compõem as regiões possíveis do mapa, assim partindo de uma mapa de adjacência ela retorna os países adjacentes àquele Country passado como parâmetro que não pertencem ao seu dono, ou seja, apenas Countries adjacentes comandados por inimigos. Esta função é usada para resolver os possíveis ataques partindo de uma região específica

```
1 export const borderCountries = (country: Countries, countries:
     CountryState, player: string, sameOrigin: boolean = true) => {
2 const borders = bdCountries[country] ? bdCountries[country] : [];
3 return sameOrigin ?
4  intersection(borders, playerCountries(player, countries)) :
5  difference(borders, playerCountries(player, countries)) ;
6 };
```

2.3.2.4 core/transictions/gameTransictions

O processo de contagem de regiões para a definição de quantas tropas estão disponíveis para o jogador é representado pela função **computeNewTroops**, que recebe um identificador de jogador e um CountryState, retornando o resultado do cálculo de suas

tropas disponíveis resultantes de tanto do seu número de territorios conquistados como o seu domínio sobre continentes.

```
export const computeNewTroops = (player: string, country: CountryState) => {
const countries = playerCountries(player, country);
let newTroopsCounter = Math.ceil(countries.length / 2);
const continentBonus = map <ContinentInfo, number> (continentsInfo, (continent: ContinentInfo) => (
 difference(continent.countries, countries).length === 0 ? continent.troopsBonus : 0
));
newTroopsCounter += continentBonus.reduce((acc, value) => (acc + value), 0);
return newTroopsCounter;
};
```

A função **gameInit** decide o estado inicial do jogo a partir do jogadores que participam da sessão, representando a parte de distribuição de territórios tradicional do jogo Risk, aleatoriamente distribuindo as regiões aos jogadores da sessão.

```

export const gameInit = (players: PlayerState) => {
each(
 object(keys(players), partition(shuffle(allCountries), lenght(players))) as InitCountries,
 (countriesNames: Countries[], playerName: string) => {
   store.dispatch(massChangeOwner(countriesNames, playerName));
 }
);
};
```

A função **endGameVerify** tem o propósito de verificar se as condições para o término do jogo estão presentes verificando o estado atual da aplicação. Para fins de simplicidade o objetivo de todos os jogadores é dominar todos os países.

```
export const endGameVerify = () => {
// Verify each player objective. For now only if a player has all countries
const { country } = store.getState();
const randomPlayer = country.Brazil.owner;
return filter(country, (countryInfo: CountryInfo) => (countryInfo.owner !== randomPlayer)) === [];
```

```
};
```

2.3.3 Currying

No código a seguir, é possível ver a utilização de currying em TypeScript e sua interface de especificação. Nesse caso não é um currying completo, já que uma das funções possui dois parâmetros. Entretanto, é necessário, caso contrário seria somente um uso forçado.

Devido a forma que a engine lida com a ligação de eventos assíncronos, é necessário deixar ligado ao escopo da função todos valores que ela precisa conhecer, já que no momento que ela é disparada o contexto de execução é outro e geralmente está ligado com a DOM. Então é gerado uma função que já está ligada com os dois parâmetros, e no momento que é dispara o browser injetado o objeto com a descrição do evento que ocorreu.

```
export interface CountryProps {
...
 onAction: ((name: Countries, type: CountrySelection) => (event: any) =
   > void);
}
  ...

   return (
     <g
       className={classNames('country-container', {'country-container-
   selectable': selectable})}
       onClick={selectable ? onAction(name, interactionState === '
   SELECT' ? 'SELECTION-OUT' : 'SELECTION-IN') : undefined}
       onMouseEnter={selectable ? onAction(name, 'HOVER-IN') :
   undefined}
       onMouseLeave={selectable ? onAction(name, 'HOVER-OUT') :
   undefined}
     >
       <defs>
         <path id={shapeID} d={shape} />
       </defs>
       {fadding}
       {country}
```

```
19          <TroopsMarker position={centroid} troops={troops} color={
        viewMode === 'CONTINENT' ? playerColor.normal : '#000000'} />
20          {troopsChanges}
21        </g>
22      );
```

2.3.4 Pattern matching

Diferente de outras linguagens que apresentam características funcionais, Typescript não apresenta suporte nativo a uma função de pattern matching com o match-with da linguagem F#, um aspecto muito importante de linguagens deste paradigma. Assim, existe a possibilidade de simular esse comportamento, mas a funcionalidade do match sendo mockado é limitada, pois necessita de um preparo prévio do ambiente. Devido a simplicidade dos casos nos testes de controle, decidimos usar apenas as ferramentas da linguagem disposta pelo professor para este caso.

2.4 Paradigma de orientação a objetos

2.4.1 Estrutura do projeto

Na pasta `src` se encontra todo o fonte do projeto com a seguinte hierarquia de pastas:

- **Components**: componentes React. Separado por componentes chaves, mas que possuem outros componentes relacionados/dependentes.
- **Core**: lógica de negócio em geral. Com as seguintes subpastas:
  - **Transducers**: funções puras, como mapeamento de estado para propriedades, levantamento de possíveis países para atacar.
  - **Transitions**: funções que precisam ler o estado da aplicação para operar.
- **Hocs**: componetes de alta ordem React.
- **Store**: gerenciador de estado Redux. Separado por grupo de funcionalidades, contendo o seu redutor e suas ações.

- **Utils**: utilidades em geral, como funções de alta ordem para manipulação de listas e objetos, cores e etc.

Existem arquivos soltos dentro da pasta `src` que são os arquivos de ponto de entrada para a criação da aplicação.

2.4.2 Classes

2.4.3 Encapsulamento

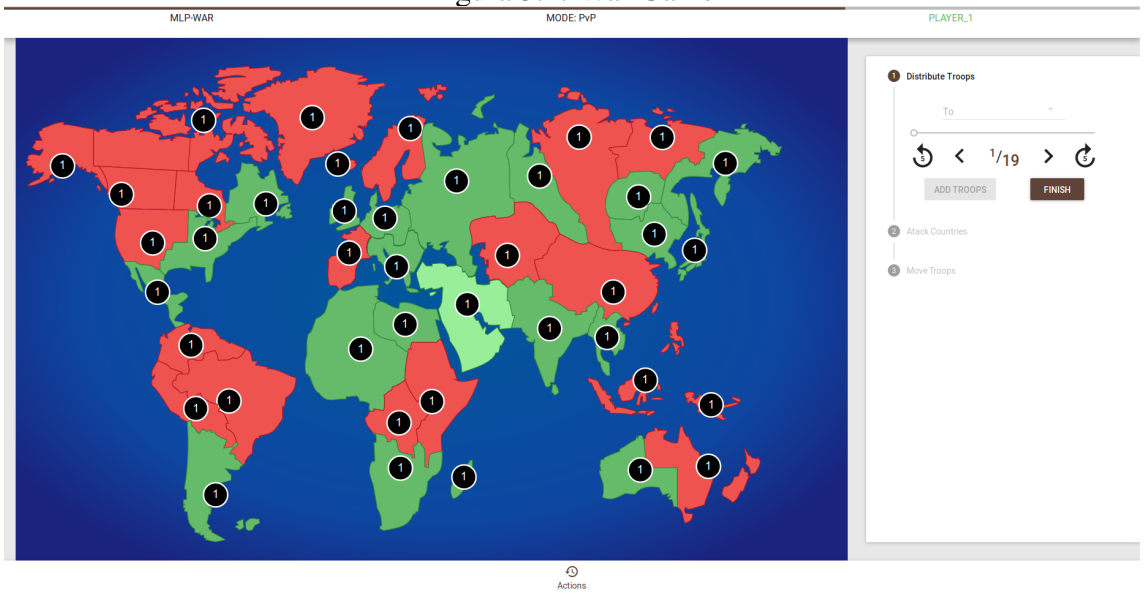
2.4.4 Singleton

2.4.5 Métodos

2.4.6 Herança

3 RESULTADOS

Figura 3.1: War Game



O link `https://bfbechlin.github.io/war-game/` fornece a página pública do projeto, através da ferramenta GitHub Pages, disponibilizada pelo próprio GitHub.

4 CONCLUSÃO

A aplicação ainda está em fase de desenvolvimento, suportando o modo jogador contra jogador e somente com o objetivo de conquista de todos os países, tanto a os objetivos quanto às trocas de cartas existentes no jogo original já estão na arquitetura necessitando de trabalho em sua interface gráfico. A interface gráfica também irá receber melhorias até o fim do processo de desenvolvimento.

Quanto ao desenvolvimento, até o momento estamos bem satisfeitos com as escolhas realizadas. As bibliotecas escolhidas tem grande sinergia com o paradigma funcional, como também a linguagem Typescript.

REFERÊNCIAS

The TypeScript Reference,
https://www.typescriptlang.org/docs/home.html

The ReactJS Reference,
https://reactjs.org/docs/react-api.html

Using Redux with React,
https://redux.js.org/basics/usage-with-react

Underscore JS,
http://underscorejs.org/

React and Redux TypeScript Guide,
https://github.com/piotrwitek/react-redux-typescript-guide

Pattern Matching with Typescript,
https://pattern-matching-with-typescript.alabor.me/